FFT Butterfly using carrysave

- **FFT butterfly equations in complex form**

  Z1 = A + W*B

  Z2 = A - W*B

- **Real and Imaginary equations**

  Z1r = Ar + Wr*Br - Wi*Bi

  Z1i = Ai + Wi*Br + Wr*Bi

  Z2r = Ar - Wr*Br + Wi*Bi

  Z2i = Ai - Wi*Br - Wr*Bi

1-1

- The first set of equations are in complex form.
  - they represent the core operation of the Fast Fourier Transformation, or FFT

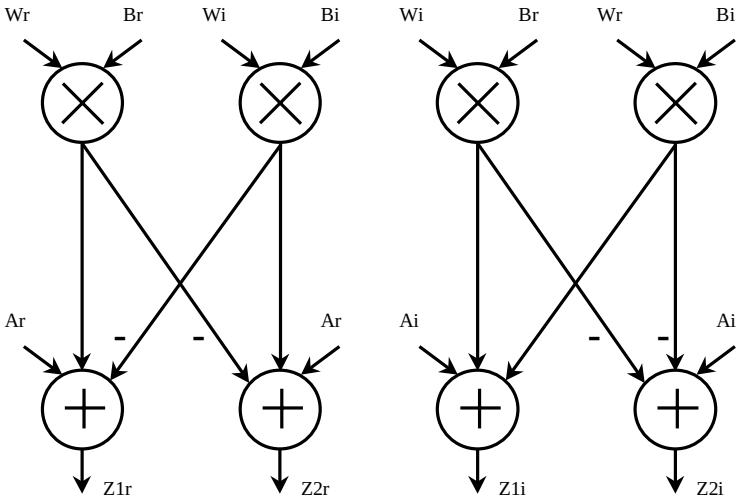
- The term butterfly refers to the symmetry in the equations.
  - this symmetry will be even more apparent on the next slide

- The second set of equations are really the same equations.
  - they have been expanded into their real and imaginary components

- Notice that there are
  - 8 multiplications
  - 4 additions
  - 4 subtractions

- In computational designs, multiplication and carry propagation are very expensive
  - we want to minimize the number of multipliers and adders
  - but we still want to run very fast

FFT Butterfly using carrysave

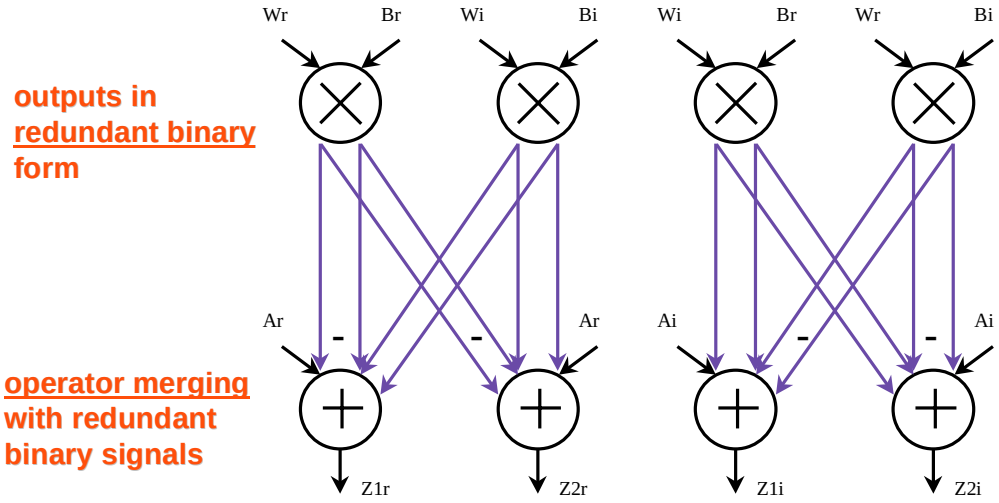


operator merging
only one carry
propagate adder

1-2

- Here the butterfly symmetry is visible in the block diagram.
- We can implement the equations with 4 multipliers.
  - each has a fanout of two

FFT Butterfly using carrysave



1-3

- Normally a multiplier has a carry propagate adder inside (this generates a binary result).

  with MC we can leave the multiplier results in carrysave format (this is a redundant binary format) this means the multiplier outputs come from the end of the multiplier's internal carrysave addition tree structure

- We will also merge the addition and subtraction for each output into a single structure.
- The result with MC is that we can implement this design with only

  4 multipliers (with carrysave outputs)

  4 carry propagate adders

- This is extremely efficient!

  both **area and delay** are minimized

  area is reduced by only having 4 multipliers AND by not having carry propagate adders inside the multipliers

  delay is reduced because each path has ONLY ONE carry propagation

FFT Butterfly using carrysave

```
// 4 product terms left in carrysave format -- no carry propagate add
directive (carrysave="on");
wire [2*w] P1 = Wr*Br;
wire [2*w] P2 = Wi*Bi;
wire [2*w] P3 = Wi*Br;
wire [2*w] P4 = Wr*Bi;

// 4 result terms in binary format -- 4 carry propagate adders
directive (carrysave="off");
wire [2*w] Z1r = (Ar<<w) + P1 - P2;
wire [2*w] Z2r = (Ar<<w) - P1 + P2;
wire [2*w] Z1i = (Ai<<w) + P3 + P4;
wire [2*w] Z2i = (Ai<<w) - P3 - P4;
```

1-4

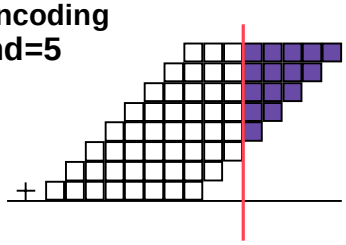
- With MCL it is very easy to directly generate signals in carrysave format.
  - the concept of redundant binary numbers is an *explicit* part of the language

- Just set the **carrysave** attribute to "on".

Internal Rounding

- **Internal rounding enables a tradeoff between computational accuracy and circuit area**
- **Simple example -- 8x8 multiplier with unsigned inputs and non-booth encoding**

  intround=5



- **Internal rounding works with any addition-based expression, e.g., A+B+C-D+ X*Y > L*M - 314*H**

1-5

•In many applications there is a complex trade-off between

computational accuracy -vs- circuit area

•Module Compiler extends this principle significantly.

•The illustration above shows the concept of *internal rounding*.

•With internal rounding, data below a user programmable bit position can be discarded.

this will reduce area, but it will also increase error

•MC automatically generates

a bit exact RTL model for simulation (the netlist is also bit exact, obviously)

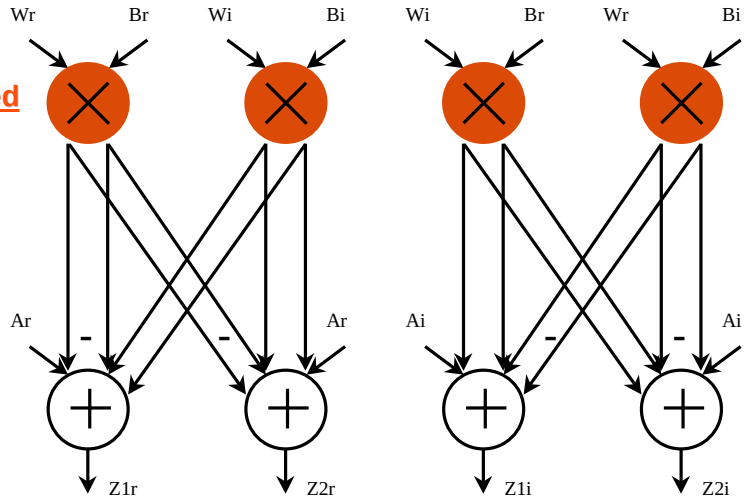
an error analysis report in the design report file

•Internal rounding can be used with **<u>any</u>** addition based expression.

A plus B plus C minus D plus X times Y ***is approximately greater than*** L times M minus 314 H

FFT Butterfly using carrysave

internally rounded multipliers



1-6

- The shaded multipliers are internally rounded

FFT Butterfly using carrysave and internal rounding

```
// 4 product terms left in carrysave format -- no carry propagate add
// programmable internal rounding position
directive (carrysave="on",intround=ir);
wire [2*w] P1 = Wr*Br;
wire [2*w] P2 = Wi*Bi;
wire [2*w] P3 = Wi*Br;
wire [2*w] P4 = Wr*Bi;

// 4 result terms in binary format -- 4 carry propagate adders
directive (carrysave="off",intround=0);
wire [2*w] Z1r = (Ar<<w) + P1 - P2;
wire [2*w] Z2r = (Ar<<w) - P1 + P2;
wire [2*w] Z1i = (Ai<<w) + P3 + P4;
wire [2*w] Z2i = (Ai<<w) - P3 - P4;
```

1-7

- With MCL it is very easy to explore this area-accuracy trade-off.

- Just use the **intround** attribute as shown above.